TITLE:

List Processing on a Data Flow Machine (Mathematical Methods in Software Science and Engineering : Third Conference)

AUTHOR(S):

AMAMIYA, MAKOTO; HASEGAWA, RYUZO; MIKAMI, HIROHIDE

CITATION:

AMAMIYA, MAKOTO ...[et al]. List Processing on a Data Flow Machine (Mathematical Methods in Software Science and Engineering : Third Conference). 数理解析研究所講究録 1981, 436: 82-116

ISSUE DATE:

1981-09

URL:

http://hdl.handle.net/2433/102759

RIGHT:

京都大学学術情報リポジトリ
KURENAI 紅
Kyoto University Research Information Repository

数理解析研究所講究録
436巻 1981年 82-116

82

List Processing on A Data Flow Machine

Makoto AMAMIYA, Ryuzo HASEGAWA,
and Hirohide MIKAMI

Musashino Electrical Communication Laboratory, N.T.T.
Midoricho 3-9-11 Musashinoshi Tokyo 180 Japan

1. Introduction

The data flow machine, whose basic idea was offered by J. B. Dennis [1] and for which several researches are pursued at several places in the world [2,3,4,5,6], is very attractive concept as a computer architecture from the following view points:

(1) The data flow machine exploits parallelism inherent in problems, and executes it in a highly concurrent manner.

(2) The recent advances in VLSI technology are noteworthy. One of the main problems of computer architecture is the way to construct the systems which utilize a large amount of VLSI devices. The data flow machine enables the implementation of distributed control mechanism which is a key problem when making use of VLSI devices.

(3) It is a serious problem in software engineering how to make a program highly productive and easy to verify, test and maintain. One solution for this problem is to write side-effect free programs based on functional programming concept. The data flow

machine effectively executes the side-effect free programs written in a functional language such as pure lisp, due to the parallelism.

(4) Non-deterministic execution will become one of the important mechanism in computer systems, when making the problem solving concept obtained in the AI research more applicable to the real world problems. The data flow machine is expected to execute non-determinism effectively due to parallel processing.

However, there exist many problems to be solved in order to make the data flow machine actual in real environment. Especially, from the view points of (3) and (4), it is necessary to make clear the applicability of the data flow machine to the non-numerical problem. This implies the necessity of solving the problem of structure memory construction. List processing is typical of non-numerical data processing. This paper discusses list processing on a data flow machine, with the Lisp data structure and operations in mind. The main reasons why Lisp is considered are that Lisp has simple and transparent structure in program and data, and that it contains the basic problems in the structure data manipulation even in the simple structure.

In conventional Lisp implementations, it is inevitable to introduce the side-effect facilities such as prog-feature in order to obtain efficient executions on von-Neumann type computers, i.e., high speed execution under the sequential execution control environment and efficient memory usage under the centralized memory management. However, if machines obtain highly parallel execution under the data flow control

environment, and if VLSI technology offers computer architecture the high level functional memory devices, side-effect free and pure functional list processing will make sense in practical use.

This paper shows that the data flow control is effective in list processing. First, the parallelism in list processing is discussed and it is pointed out that it can be achieved by parallel evaluation of function arguments and partial execution of function body. Then it is shown that the parallelism increases dramatically by introducing lenient cons concept into the data flow execution control; that is, the lenient cons brings data flow systems the pipelined execution among activated functions because each activated function executes each data item (in list) as soon as it is partially constructed. The effect of lenient cons on list processing is shown through analyses of several programs.

Finally a garbage collection algorithm based on the reference count method is discussed. The garbage collection is also important problem in list processing in order to utilize the memory cells effectively. The algoritm is essentially parallel in the sense that cells are reclaimed whenever it becomes useless, independently of the foreground list manipulation.

All programs throughout this paper are described in the language Valid [7] which is designed as a high level programming language for the data flow machine presented in this paper.

2. List processing under the data flow control environment

The remarkable effects of data flow execution control are the followings.

(1) It exploits maximal parallelism inherent in a given program both on a low level (primitive operation level) and on a high level (function activation level).

(2) It effectively executes programs constructed based on the concept of functional progamming which has no notion of program variables and side effects (i.e., re-writing the global variables).

The parallelism of primitive operation level is obtained by the data driven control principle; that is, each operation is initiated without attention to other operations when all of its operands have arrived. The parallelism of function activation level is obtained by the partial evaluation mechanism: (a) Each argument of a function is evaluated concurrently. (b) The execution of a function is initiated when one of the arguments of the function is evaluated, and the caller function resumes its execution when one of the return values is obtained in the invoked function execution.

In conventional list manipulation, there exists a case in which the partial execution of function evaluation is not effective, since an execution using a list data which is generated by another function has to wait until its construction is completed. However if lenient cons concept is introduced, the consumer function which uses list elements partially can start

its execution as soon as the producer function generates a fragment of the list partially before it constructs the whole list.

In this section, these parallel execution mechanisms are examined through several examples, whose programs are written in Valid. (Some of Valid features are described in Appendix A.)

2.1 Parallel evaluation of arguments

Programs written in Valid are equivalently transformed to pure functional representation, i.e., the form of prefix notation, and equally translated to data flow graphs. For instance, Programl which reverses a given list in each level is translated by the Valid compiler into the data flow graph shown in Fig.2.1. Block2 in Programl is equivalently represented in the prefix notation

fulrev(cdr(x), cons(fulrev(car(x), nil), y)) .

In this expression, the two arguments cdr(x) and cons(...) of the function fulrev are evaluated in parallel, and before evaluating the argument cons(...) , its two arguments fulrev(...) and y are evaluated in parallel, and so on. Thus the evaluation of a function, in general, proceeds from the inner to the outer (i.e., innermost evaluation), and this results in the highly parallel evaluation of the innermost arguments. This is equivalently said in other words; each evaluation is independent of the other evaluations under the condition that the evaluation is initiated only when the values of all of the arguments are

obtained, (which is called a data-driven control).

```
Programl -- Mirror image of tree

fulrev: function (x,y) return (list)
= case
   null(x) -> y;
   atom(x) -> x;
   others  -> clause
                u = cdr(x);
                v = fulrev(car(x),nil);
    block2      w = cons(v,y);
                return fulrev(u,w)
              end
  end
```

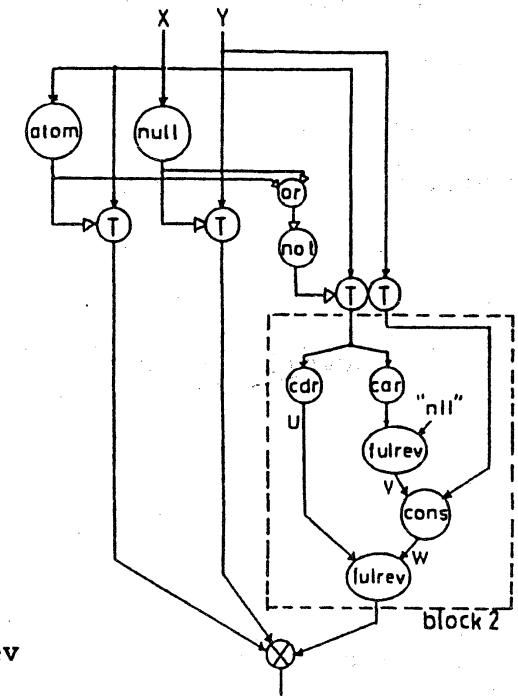


Fig.2.1
Data flow graph of fulrev

The parallelism in Lisp 1.5 interpretation [8] is also achieved by implementing the interpreter based on the data flow control concept. An implementation is represented in Appendix B. In the implementation, the parallel execution is exploited in the evcon and eval recursion. In the course of argument evaluations, evlis forks eval function for each argument and each evaluated argument value is consed finally. The parallel eval activation in evlis is effective. The following program written in m-expression uses divide and conquer algorithm for factorial calculation.

```
fact[m;n]
= [equal[m;n] -> n;
   t -> times[fact[m;quotient[plus[m;n];2]];
              fact[addl[quotient[plus[m;n];2]];n]]]
```

Evcon also has a possibility for parallel execution. The implementation in Appendix B uses loop control according to the Lisp 1.5 semantics in which the predicate parts of cond expression are evaluated sequentially. However if evcon is implemented by recursion as shown below, eval which evaluates the predicate part is activated in parallel.

```
Evcon: function(s,c,a) return(list)
 = if c=nil then novalue
           else clause
                 if Eval(caar(c),a) then Eval(cdar(c),a)
                                    else novalue;
                 Evcon(s,cdr(c),a)
               end
```

Where a signal value is used for the top level activation of evcon; Evcon('s',c,a).

In order to make effective use of parallel control in evcon, the conventional Lisp program must be modified. For instance, the conventional equal function is modified to

```
equal[x;y]
= [and[atom[x];atom[y]] -> eq[x;y];
   not[or[atom[x];atom[y]]] -> and[equal[car[x];car[y]];
                                   equal[cdr[x];cdr[y]]];
   t -> f]
```

Parallel execution of evcon has the potential to non-deterministic control [9], though the incarnation of mechanism and integration to semantics of Valid are the problems to be solved.

2.2 Partial execution of function body

The parallelism based on the parallel evaluation of

arguments of each function is limited because the nesting of arguments is limited in source text. This restrictions on parallelism, however, can be overcome by executing function body partially.

If the data-driven control principle is applied to the function activation, as in the case of primitive operations, every function is activated only after all of its arguments are evaluated. In this case, the unnecessary time is wasted in each function activation by waiting for the completion of all its argument evaluations. On the other hand, the partial execution of function body enables each value of arguments to be passed into the function body immediately when it is evaluated, and the execution of the body to proceed partially every time the value is passed into. In the same way, it enables each of the return values to be passed back to the calling function as soon as it is generated, and the calling function to resume and proceed the execution partially every time the return value is passed back from the called function. Where each function is permitted to return multiple values (i.e., the tuple of values) under the data flow control environment.

This mechanism has an effect that each function call has been replaced by its body initially, and that the parallel evaluation of arguments goes beyond the barrier of function invocation and restrictions mentioned above.

For instance, the function partition is activated when the argument cdr(x) or y is evaluated in the function sort of Program2, which sorts a list x by Quicksort algorithm. In the

case of this program, as the evaluation of y means the evaluation of the expression list(car(x)) , the value of cdr(x) will be generated just before the y is generated. So that, when the second argument is passed into the partition body, then-part or else-part will be ready to run. The function partition returns three values. In the else-part of the partition body, for instance, the values w1, w2 and w3 are generated by the function partition and are used in return expression, so if each of w1, w2 and w3 is returned immediately when it is generated, each of these values is used and passed back to the calling function; for example, in case of x1=y1, w1 and w3 are passed back to the calling function directly, while w2 is appended to list(x) to generate a new list which is passed back. The innermost recursion phase of partition returns each of three values nil, y, and nil, independently.

Program2 -- Quicksort program

```
sort: function (x) return (list)
= if x=nil then x
          else clause
                 y = list(car(x));
                [y1,y2,y3] = partition(cdr(x),y);
                return append(sort(y1),append(y2,sort(y3)))
               end;

partition: function (x,y) return (list,list,list)
= if x=nil then (nil,y,nil)
          else clause
                [w1,w2,w3] = partition(cdr(x),y);
                x1 = car(x); y1 = car(y);
                return
                 case
                  x1=y1 -> (w1,append(list(x1),w2),w3);
                  x1<y1 -> (append(list(x1),w1),w2,w3);
                  x1>y1 -> (w1,w2,append(list(x1),w3))
                 end
              end;
```

In this way, many partition functions are activated and are executing their body partially. As the same may be said of the function sort, Program2 sorts the input list x in the highly parallel execution.

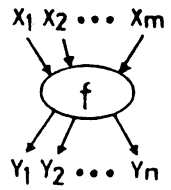
The partial execution of function body has also an effect on the Lisp 1.5 interpreter, because the interpreter executes Lisp programs in high parallelism due to the partial execution of each body of the function eval, evlis, and evcon.

The function activation and argument passing mechanism for the partial function execution is implemented as shown in Fig.2.2. The data flow graph of Fig.2.2(c) represnts the activation control of the function

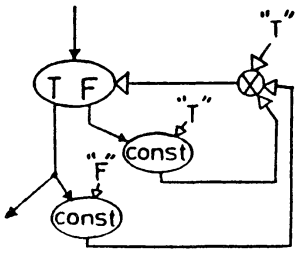
[yl,y2,...,yn] = f(xl,x2,...xm)

The copy node, which creates an new environment for the activated function (or makes a new copy of the body in logical), is initiated by the or-gating nodes when one of the tokens (values) has arrived. The orgate implementation uses t/f switch as shown in Fig.2.2(b). When the new environment is created and the body is ready to run, the token "in" (instantiation name of the activated function) is sent to link nodes and rlink nodes. Each link node passes each argument value xl, x2, ..., xm to the body of the activated function every time each value has arrived. Each rlink node passes information of the place where the return value is sent to. This information yl', y2',..., yn', each of which is determined at compilation time corresponding to yl, y2, ..., yn, are attached to each return value to identify its

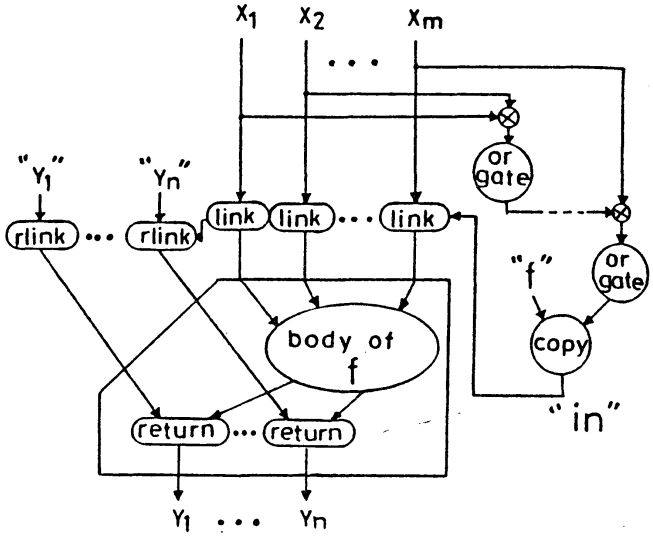
destination.



(a) Function invocation node



(b) Orgate implementation



(c) Function activation control

Fig.2.2 Fuction activation mechanism

2.3 Lenient cons and parallelism by pipelined processing

Although the partial execution of function induces higher parallelism, it is not sufficient for maximally exploiting the parallelism inherent in the given program. For instance, Program2 is expected to execute in a highly parallel fashion among activated functions of sort and partition. This parallelism, however, does not work well for reducing the execution time in the order, because the time spent to sort the list of length n is proportional to the square of n in the worst case. (Though it is proportional to n in the best case.) The reason is that; as each of the value y1, y2 and y3 is not returned until the append operation is completed in the partition body, the execution of sort function which uses those values must

wait until they are returned, and the waiting time is proportional to the length of the list data made by the append operation.

The Lisp interpreter is another example which shows that the parallelism is not maximal. As the operation of the apply to each evaled value which is returned from the function evlis must wait until all of the evaled values are constructed to a list by the cons operation which resides in the last part of the evlis body, the execution of lisp function body can not proceed partially.

If the former parts of the list which are partially generated are returned in advance during the latter parts are appended, the execution which uses the former parts of the list can proceed. Thus the executions of the producer and the consumer overlap each other. As the append is the repeated operations of cons as Program3 shows, this problem can be solved by introducing leniency into the cons opreation.

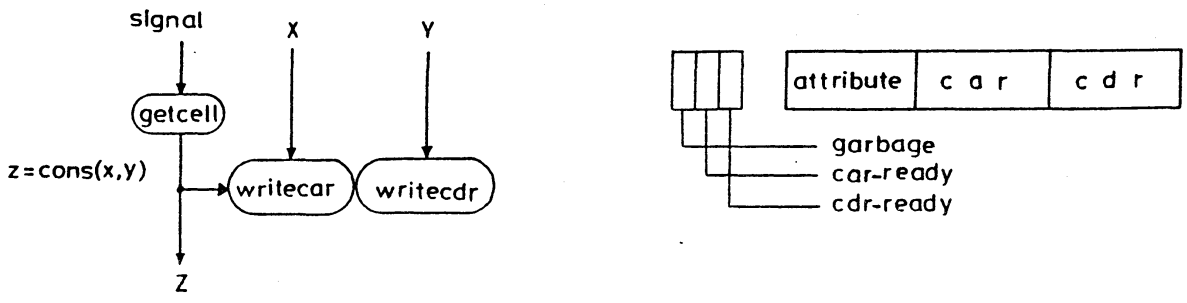
Program3 -- append

```
append: function (x y) return (list)
= if x=nil then y
          else cons(car(x),append(cdr(x),y))
```

The lenient cons, which is slightly different from the idea of "suspended cons" [10], means the following. For the operation of cons(x,y), the cons operator creates a new cell and returns its address as a value in advance before its operand x or y arrives. Then the value x and y are written in the car field and the cdr field of the cell respectively, when each of them has

arrived at the cons node. In the implementation, the cons operator is decomposed into three primitive operators, getcell, writecar, and writecdr as shown in Fig.2.3. The getcell node is initiated on the arrival of a signal token, which is delivered when the new environment surrounding the cons operation is created. The getcell operator creates a new cell, and sends its address to the writecar node, the writecdr node, and the nodes waiting for that cons value. Each memory cell has, in addition to the garbage tag, the car-ready tag and the cdr-ready tag each of which controls read accesses to the car field and the cdr field. The getcell operator resets the both ready tags to inhibit read accesses. The writecar (or writecdr) operator writes the value x (or y) to the car field (or cdr field), and sets the ready tag to allow read accesses to the field.

The lenient cons has a great effect in list processing. It naturally implements the stream processing feature in which each list item is processed as a stream [4,11] for programs which are normally written without the notion of stream according to list processing concept.



(a) Cons mechanism (b) Data cell structure

Fig.2.3 Lenient cons implementation

3. Evaluation of the lenient cons effect

The parallelism enhanced by the effect of pipelined processing which lenient cons brings about is analyzed so as to estimate the lenient cons effect. The sieve algorithm to find prime numbers and the Quicksort algorithm in chapter 2 are taken as examples.

The followings are assumed in the evaluation of parallel algoritnm:

(1) There exist an infinite number of resources, namely the operation units (processors) required for calculations are available at any time, and the time for resource allocation and the network delay is ignored.

(2) The execution of any operation completes in a unit time.

(3) The time required for function linkage is ignored.

3.1 Sieve program to find prime numbers

Program4 finds prime numbers using the sieve method of Eratosthenes. Each cons operation appeared in this program is implemented with lenient cons. The cons operation is initiated by a signal s when a block which contains the cons operation is opened.

Primenumber(n) obtains the sequence of prime numbers by sieving the sequence (2 3 ... n) generated by intseq(2,n). The lenient cons enables intseq to send out the sequence of numbers (2 3 ... n) one after another. To illustrate it more clearly, a data flow graph of intseq(m,n,s) is shown in Fig.3.1. Where s

is a signal value name which does not appear in source text but is generated by compiler.

When the function intseq(m,n,s) is initiated, if m is not greater than n the getcell node is fired and the new cell address obtained is returned immediately. Then the value m and the value of intseq(m+1,n,s) is written into the car and cdr part of the new cell respectively when each of them has arrived. Consequently, intseq sends out new numbers one by one every time it is initiated. Sieve(n,s) holds the first (car(n)) of the sequence sent from intseq, and invokes the function delete to remove the numbers in the rest (cdr(n)), which are divisible by car(n). The function delete sends out the elements indivisible by car(n) one by one, which in turn are passed to the sieve function recursively. In this way, the executions of intseq, sieve and delete are overlapped.

Table 1 traces the invocations of intseq, sieve and delete, and depicts how the algorithm works. Each row gives the sequence of numbers generated in each invocation. The intseq generates the sequence of numbers (2 3 ... n) one by one in a unit time. The top level sieve is initiated immediately when the first element of the sequence is returned from the intseq. After the activation, the sieve immediately invokes the $delete_1$, where the suffix represents invocation sequence number.

The $delete_1$ takes the sequence (2 3 ... n) from the intseq, and deletes the numbers which are divisible by 2. The $delete_i$ which is initiated in the i-th level sieve takes the output sequence of the $delete_{i-1}$, and deletes the numbers which are

divisible by the i-th prime number. The deletel returns values one unit time after the intseq, as seen from the table.

As the time needed for invocation of the delete in the sieve is constant and the $delete_i$ returns values one unit time after the $delete_{i-1}$ returns the value, the sieve returns the ith prime number i units time after the intseq returns that number.

Since there are n/log n primes asymptotically in {2,3,...,n}, the computation time needed in this program is n+n/log n, i.e. the order of n.

Program4 -- Sieve program

```
primenumber: function (n) return (list)
= sieve(intseq(2,n))

sieve: function (n) return (list)
= if n=nil then nil
          else cons(car(n),
                    sieve(delete(car(n),cdr(n))))

delete: function (x,n) return (list)
= if n=nil then nil
          else if remainder(car(n),x)=0
               then delete(x,cdr(n))
               else cons(car(n),delete(x,cdr(n)))

intseq: function (m,n) return (list)
= if m>n then nil
         else cons(m,intseq(m+1,n))
```

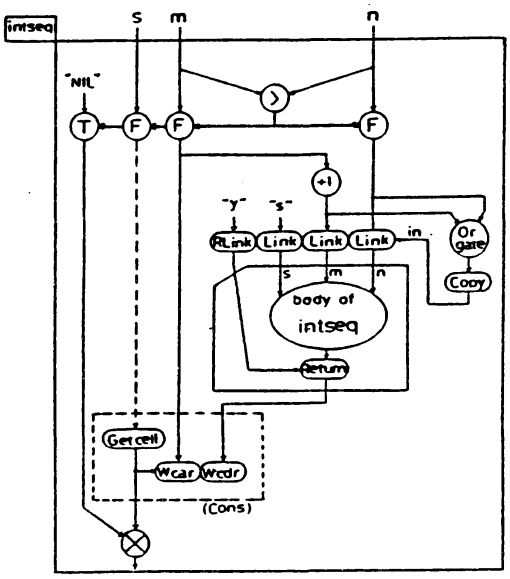
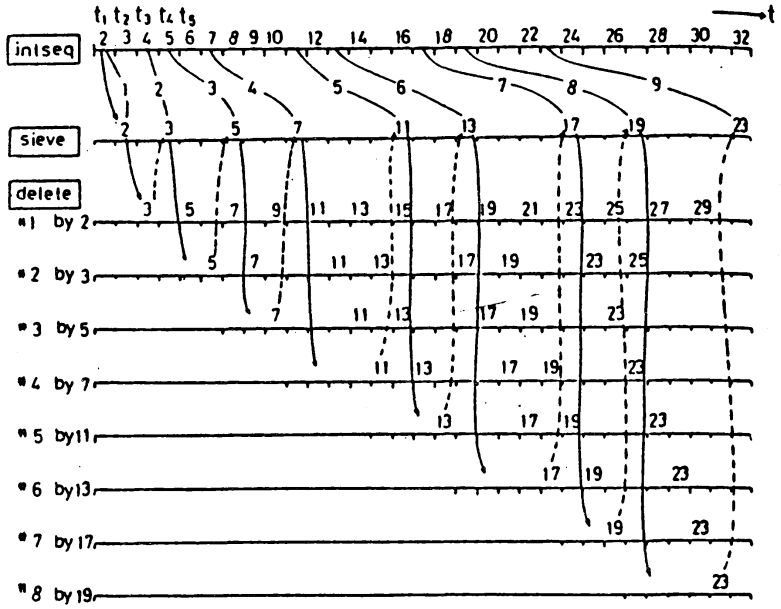


Fig.3.1 Data flow graph of intseq

Table 1 Execution of primenumber(2,n)



3.2 Quicksort program

This section analyzes the computation time of the quick sort program described in Section 2.3. To simplify the analysis, the worst case behavior of the algorithm is investigated. (Where the worst case means the case in which input data are in the reverse order.) It is assumed that the time required for the append operation is propotional to the length of a list to which another list is appended. Table 2 traces the function invocation sequence to illustrate the action of sort((3 2 1)).

Let y be (x1 x2 ... xn), where $x_i > x_{i+1}$ for i = 1, 2, ..., n-1. The function sort(y) acts as follows. The activated

function partitionl partitions a list into three lists yll, yl2, and yl3 each of which contains the elements less than, equal to, and greater than xl, respectively. For the input data of the worst case yll is (x2 x3 ... xn), yl2 is xl, and yl3 is nil. The result of sort(y), the top level activation of the sort, is obtained by appending the result of append(yl2, sort(yl3)) to the result of sort(yll). Due to the lenient cons effect, the function sort(yll) is invoked immediately after the first element of yll, i.e. x2, is obtained. The values yl2 and yl3 are returned from the partitionl 2N(yll) units time after the first value of yll is returned from that function, since it takes N(yll) units time for the first element of y (=xl) to be passed into the innermost recursion phase of the partitionl where yl2 (=(xl)) and yl3 (=nil) is generated, and it takes the same time for those values to be returned. (where N(yi) means the number of elements in yi.)

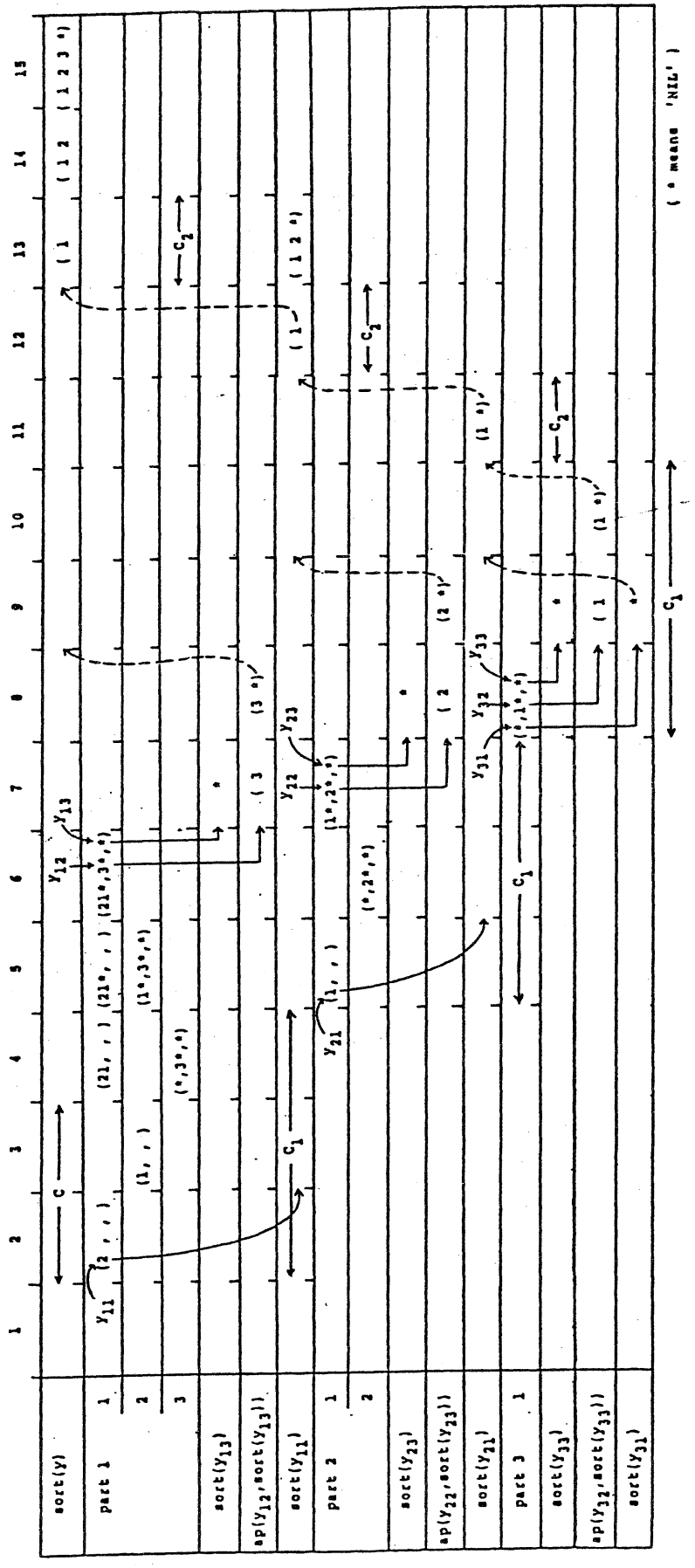
Then, in the computation of sort(yll), the second level activation of the sort, the function partition2 generates three values y2l, y22, and y23. (y2l=(x3 x4 ... xn), y22=x2, y23=nil.) Due to the lenient cons effect, the partition2 returns the first element of y2l, i.e. x3, one unit time after the partitionl returns the element x3. As it takes C (a constant, which equals 2 in this case) units time for each activated partition to generate the second element after generating the first element, the first element of y2l is returned Cl (=C+l) units time after the first element of yll is returned. Since yl2 and yl3 are returned at time 2N(y)=2n, and y22 and y23 are

returned at time C1+2N(y11)=2n+C-1, y22 and y23 are returned C-1 units time later than y12 and y13. In the same way, the partition3 which is invoked in the computation of sort(y21) returns the result y31 (=(x4 x5 ... xn)) C1 units time after the partition2 returns the result y21, and returns the results y32 (=x3) and y33 (=nil) C-1 units time later than the results y22 and y23, and so on.

Since the n partition functions are activated for sorting n elements, it takes n(C1) units time until the last partition completes. Each result of the partitions has to be constructed to a list by the append operation. Since it takes n units time for the first element to be returned from the top level append, and it takes n-1 units time for the append operation to be completed, the total computation time for sorting n elements is C1n+2n-1, i.e. the order of n.

In the best case, the total computation time is the order of log n, because the partitioning operation reduces the length of each list to the half, and the invocation tree of the sort function is balanced with the height of log n; that is, the execution acts on the divide and conquer strategy.

Table 2 Example of the worst case -- sort(3 2 1)



4. Garbage collection

As many data are copied, used, and thrown away very often in the course of the side-effect free data manipulation, it is very important to resolve the problem of how to utilize structure memory cells effectively, namely the effective garbage collection method under the parallel processing environment.

Although mark-scan methods are generally used as a garbage collection method in the conventional machine, the reference count method is adopted here. The reason is that; (a) Since the data tokens which are pointers to list data entries in the structure memory are scattered in the various parts of the machine such as instruction memory units, communication networks and operation units [11,12], it is very difficult to extract the active cell without suspending the execution. (b) As the list manipulations have no side effect and every list is made only by the cons operation, no circular lists are created.

This section presents a fundamental method, though it is not efficient, in order to introduce the basic idea at first, then describes a revised method which is much more efficient.

In the method presented here, each data cell has the reference counter field which is updated every time the data cell is accessed. The reference counter handling algorithm in the fundamental method is written in Algol-like language for each primitive list operation in the following. Where r(x), z and d denote the reference count field of the cell x, the value of each operation, and the number of operation nodes which are waiting

for the value z, respectively.

```
procedure Car(x,d);
  begin
      z := car(x);
      Red(x);
      r(z) := r(z)+d
  end

procedure Cdr(x,d);
  begin
      same as Car(x,d)
  end

procedure Atom(x);
  begin
      z := atom(x);
      Red(x)
  end

procedure Eq(x,y);
  begin
      z := eq(x,y);
      Red(x);
      Red(y)
  end

procedure Cons(x,y,d);
  begin
      z := cons(x,y);
      r(z) := d
  end

procedure Red(x);
  begin
      r(x) := r(x)-1;
      if r(x)=0 then begin
                         Red(car(x));
                         Red(cdr(x))
                     end
  end
```

One problem of completeness in the reference count method occurs in the execution of a conditional expression. In the case of if p then f(x) else g(y) , for example, as g(y) is never executed when p is true, the reference count of cell y is left un-decremented, and as a result, the cell y is never reclaimed

though it is a garbage in virtual. In order to avoid this, a special operator erase, which executes only the procedure Red, is prepared. (Fig.4.1)

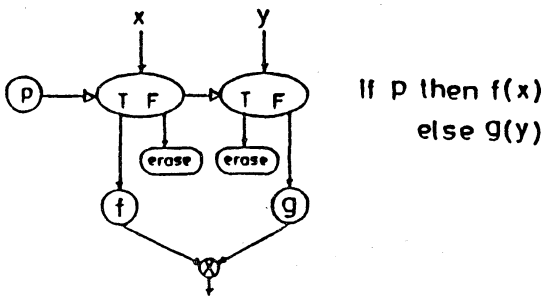


Fig.4.1 Conditional expression and erase operation

The reference count handling overhead is serious because updating the reference count is needed in all operations, not only in the five primitive operations but in the switch and gate operations as shown in the case of conditional expression. The number of reference count updating can be reduced by eliminating the redundant updating if it is possible to make use of the structure in high level language, i.e. the block structure and scope rule.

In stead of implicitly updating the reference count in every operation, the revised method explicitly updates it by using the increment and decrement operators which the Valid compiler generates by making use of several Valid features which are suitable for eliminating the redundant reference count handling. The reference number of a cell indicates the number of value names which denote the cell in the program text. Where the value names are explicitly defined in the value definition or implicitly defined in such cases as cons values and function

values. The reference number of the cell which is newly denoted in a block is incremented when the block is opened and decremented when closed. As all value names are defined uniquely in a block and are local to the block defining them, due to the features of Valid, each value name refers only one cell and the cell is never referred by the value name outside the block. Therefore the reference number is incremented only once even if a number of operations refer the value within the block.

A Valid source program fragment, for instance,

```
[x,y] = clause
          x = E1; y = E2; z = E3;
          return(E(x,y),x)
        end;
```

is compiled to the data flow graph shown in Fig.4.2. Where E1, E2, E3, and E represent expressions. The return expression generates two values which are implicitly denoted by ret1 and ret2 then explicitly denoted by x and y in the environment outside the block. The reference number of local values x, y and z should never be decremented before the reference number of return value ret1 and ret2 are incremented, so as to prevent the cells pointed by the return values from being reclaimed during the transient time of return value passing. The and-gating node and gating nodes keep the order of increment and decrement operation safe for garbage collection.

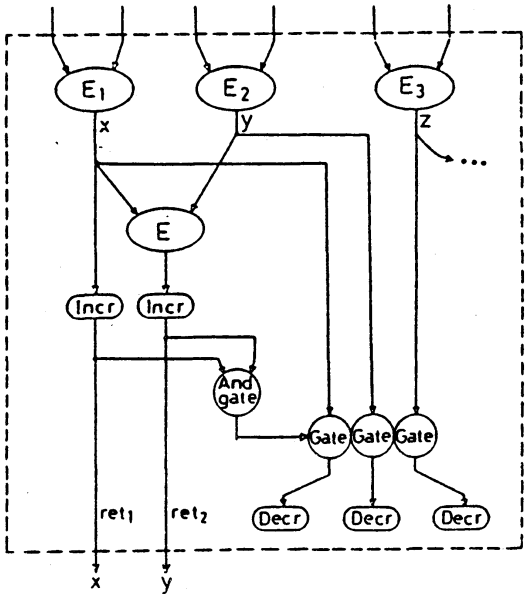


Fig.4.2

Reference count management of local value name

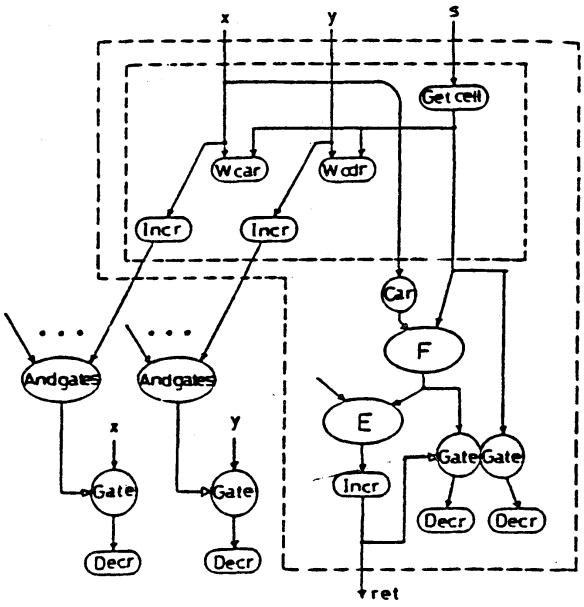


Fig.4.3

Reference count management in the expression with cons operation

Even if they are not denoted explicitly but are obscured in expressions, values of cons operations or function invocations have to be treated as denoted in order to prevent cells created by cons operation or function body from being reclaimed. So the expression containing cons operation or function invocation as sub-expressions is interpreted as being composed of the pseudo blocks. For example, the expression

E(...,f(cons(x,y), car(x)), ...)

is interpreted as

```
clause
  u' = clause z'=cons(x,y); return z' end;
  v' = f(u',car(x));
   . . .
  return E(...,v',...)
end
```

and compiled to the graph shown in Fig.4.3. Where f represents a function invocation. The readers should note that the getcell operator initializes the reference number of the created cell to one, and the function body, when it returns values, increments the reference number of the cell pointed by the returned value.

Another problem on safety arises in the case of lenient cons. As for the above example, if the value x or y is pointed to by the cell newly created in this block, its reference count must not be decremented before it is made sure that the increment operation of each reference count is completed, since, through the write-car or write-cdr operation, each of the cells is to be pointed by the cell created by getcell operation.

The general rule for the cons operation to guarantee the safeness in garbage collection is :

In the cons operation

$$\text{cons}(u,v) \qquad u \equiv E_x(x1,x2,\ldots,xn)$$
$$v \equiv E_y(y1,y2,\ldots,ym) \ ,$$

where E_x and E_y are expressions composed of value names xl,...,xn, and yl,...,ym each of which is a name denoting each value explicitly or implicitly (; namely, cons or function

invocation is replaced by xi or yi), the decrement operation of the reference count of each cell pointed by xl, ..., xn, yl, ..., and ym is postponed until the increment operation of the reference count of the each cell pointed by u or v is completed.

5. Conclusions

This paper discussed some issues in list processing under the data flow control environment from the view point of parallelism. The basic philosophy of the data flow machine architecture presented in this paper is that the highly parallel execution is achieved by the data flow control concept both on the primitive operation level and on the function activation level. The mechanism of partial execution in each function was shown to be effective to exploit the parallelism in list processing through some examples including an implementation of lisp interpreter, then the lenient cons mechanism was shown to exploit the parallelism maximally, through the analysis of two programs, parallel sieve program to find prime numbers and Hoare's quick sort program, which proved each execution is the order of linear time.

Then the garbage collection mechanism based on the reference count method was described. The algorithm works well as a parallel garbage collection algorithm in the sense that the garbage is reclaimed all the time, concurrently with the foreground list operations.

Although the architecture of the data flow machine was

omitted in this paper, the characteristics of the machine , which is based on associative memory and logic-in-memory concept, are that the machine is composed of many modules so as to realize the parallelism which can be exploited logically. The details are described in [10,11].

There remains many problems to be solved in order for the machine to be available in practical use. Several works are in progress to examine the effectiveness of the theory described in this paper. These include the construction of software simulator, the design of experimental hardware system, and the implementation of Valid compiler. The simulator, the first version of which is now running, is to collect the statistical data which exhibit the effect of lenient cons, the effect of cons strategy and memory partition, the garbage collection overhead, etc. The design of experimental hardware system offers the data to estimate the cost performance. The Valid compiler which generates the object code for the simulator and experimental hardware is a tool for establishing the programming system based on the functional programming methodology, on the data flow machine.

References

[1] Dennis,J.B., "A Preliminary Architecture for a Basic Data Flow Processor", The Second Annual Symposium on Computer Architecture, Jan., 1975, pp.126-132.

[2] Plas,A., "LAU System Architecture: A Parallel Data-Driven

Processor Based on Single Assignment", Proceedings of the International Conference on Parallel Processing, 1976, pp.293-302.

[3] Watson,I. and J.Gurd, "A Prototype Data Flow Computer with Token Labelling", AFIPS Conference Proceedings 48, 1979, pp.623-628.

[4] Arvind, K.P.Gostelow and W.Plouffe, "An Asynchronous Programming Language and Computing Machine", Report TR 114a, Department of Information and Computer Science, University of California, Irvine, California, December, 1978.

[5] Davis,A.L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", Proceedings of the Fifth Annual Symposium of Computer Architecture, Apr., 1978, pp.210-215.

[6] Keller,R.M., G.Lindstrom and S.Patil, "An Architecture for a Loosely-Coupled Parallel Processor", UUCS-78-105, University of Utah, Salt Lake City, Utah, 1978.

[7] Amamiya,M., "A Design Philosophy of High Level Language VALID for Data Flow Machine", Proceedings of IECEJ Annual Conference, 1981, NO.1486, in Japanese.

[8] McCarthy,J. et.al., "Lisp 1.5 Programmer's Manual", MIT Press, Cambridge, Massachusetts, 1960.

[9] Dijkstra,E.W., "Guarded Commands, Non-determinacy, and Formal Derivation of Programs", Comm.ACM, 18, 8, 1975, pp.453-457.

[10] Friedman,D.P. and D.S.Wise, "CONS should not evaluate its arguments", Automata, Language and Programming, S.Michaelson and R. Milner, Eds, Edinburgh Univ. Press, 1976.

[11] Dennis,J.B. and K.S.Weng, "An Abstract Implementation for Concurrent Computation with Streams", Proceedings of International Conference on Parallel Processing, 1979, pp.35-45.

[12] Amamiya,M., R.Hasegawa and H.Mikami, "A List Processing Oriented Data Flow Machine Architecture", Proceedings of Meeting on Symbol Manipulation, IPSJ, 13-3, 1980, in Japanese.

[13] Amamiya,M., R.Hasegawa and H.Mikami, "List Processing Oriented Data Flow Machine and Its Software Simulator", Proceedings of Meeting on Computer Architecture, IPSJ, 40-8, 1981, in Japanese.

Appendix A. Brief description of Valid language features

Valid is a high level language designed for programming on a data flow machine. The name "Valid" is the abbreviation of VAlue IDentification language. The basic philosophy of design is that Valid semantics supports programming based on functional concept, while its syntax offers the conventions to write programs in Algol style. This appendix describes some features which helps readers to understand sample programs in this paper.

The characteristics of Valid are as follows.

(1) The basic structure of Valid is expressions and definitions, i.e. Valid has no concept of program variables and assignments, instead the values are denoted by value names, if necessary. This concept is different from single assignment concept in the sense that variables only mean the names which are assinged to each value.

(2) A Valid program consists of function definitions, macro definitions and value definitions. A function (or macro) definition defines a function name and its body which returns multiple values. The body is an expression. Functions are invoked in the execution time by their names, while macros replace their names by their bodies at the compilation time. Value definitions also define multiple values. For example, the value definition

[x1,x2,...,xn] = expression

defines n values generated by expression and denotes them by value names x1, x2, ..., xn.

The function definition

f: function (a1,a2, ..., an) return (b1, b2, ..., bm)
= expression

defines the function f which has formal arguments (value names) a1, a2, ..., an and returns values of type b1,b2, ...,bm.

(3) Valid has a block structure. A block is a set of definitions. In a block all value definitions are sequence free, namely each value definition is evaluated in parallel within the block. In Valid, the delimiter semicolon (;) which delimits value definitions is used in the different meaning from Algol. The semicolon (;) separates two constructs in parallel, i.e. A;B is identical to B;A, while the delimiter comma (,) separates two constructs keeping the sequence, i.e. A,B is not same as B,A.

Functions and macros may be defined in any block locally.

All function names, macro names and value names can be refered from other places in the program text under the constraints of scope rule as in Algol; i.e., every identifier denotes a local value which may be refered within a block surrounding its definition. A block becomes an expression, and value(s) of the block is generated by a return expression occurring in the block.

(4) All iterations are described based on the recursion. The conventional loop is described in the recurrence expression of the following form.

for (<iteration value names>) : (<initial values>)
do <block including recur expression>

Other features of Valid as a general purpose high level language whose explanation is omitted because they have nothing to do with the discussion of this paper includes;

(5) Valid permits several data types. Those include Boolean, integer, real, array, record, list, and signal. Each value is specified with its type name (if necessary). Since this paper deals with only the list type, type specification is omitted.

(6) Parallel expression based on fork-join concept which is powerfull for describing blocks executable parallelly such as vector operation.

Appendix B. Data flow implementation of Lisp 1.5 interpreter

Though each function of Lisp 1.5 interpreter is defined as a

recursive function, it is necessary to reconstruct those functions so as to realize the efficient execution control as in the case of the conventional von Neumann machines. The functions implemented from this view point on the data flow machine are described in Valid. The implementation is based on the principles of that: (1) The tail-recursive structure, which has no parallelism, is implemented by the loop control so as to reduce the function invocation overhead. (2) The parallelism is exploited only from the non tail-recursive structure. For example, the functions apply and eval are implemented by the loop in which programs apply1 and evall execute alternatively, while the function evlis which has non tail-recursive structure forks eval functions.

Program5 -- Lisp 1.5 interpreter

```
Apply: function (fn,args,alist) return (list)
= for (x,y,z):(fn,args,alist)
   do clause
      [t,xx,yy] = Apply1(x,y,z);
      xxx = if t then xx
            else clause
                 [t1,x1,y1,z1] = Evall(xx,yy);
                  x2 = if t1 then x1
                                else recur(x1,y1,z1);
                 return(x2)
                 end;
      return(xxx)
      end;
```

```
Apply1: macro (fn,args,alist) return (list,list,list)
= for (x,y,z):(fn,args,alist)
    do case
        atom(x) -> case
                    x='car' -> return('t',caar(y),nil);
                    x='cdr' -> return('t',cdar(y),nil);
                    x='cons' -> return('t',cons(car(y),cadr(y)),nil);
                    x='eq' -> return('t',eq(car(y),cadr(y)),nil);
                    others -> recur(Eval(x,y),y,z)
                   end;
        car(x)='lambda' -> clause
                            x' = caddr(x);
                            y' = Pairlis(x,y,z);
                            return(nil,x',y')
                           end;
        car(x)='label' -> clause
                           x' = caddr(x);
                           z' = cons(cons(cadr(x),caddr(y)),z);
                           recur(x',y,z')
                          end
       end;

Eval: function (e,a) return (list)
= for (x,y):(e,a)
   do clause
       [t,z1,z2,z3] = Evall(x,y);
       xx = if t then z1
                 else clause
                       [t1,y1,y2] = Apply1(z1,z2,z3);
                       if t1 then y1 else recur(y1,y2)
                       end;
       return(xx)
      end;

Evall: macro (e,a) return (list,list,list,list)
= for (x,y):(e,a)
   do
     case
       atom(x) -> return('t',cdr(Assoc(x,y),nil,nil);
       car(x)='quote' -> return('t',cadr(x),nil,nil);
       car(x)='cond' -> clause
                         [x',y'] = Evcon(cdr(x),y);
                         recur(x',y')
                        end;
       others -> clause
                  y1 = car(x);
                  y2 = Evlis(cdr(x),y);
                  y3 = y;
                  return(nil,y1,y2,y3)
                 end
     end;
```

```
Evconl: macro (c,a) return (list,list)
= for (x,y):(c,a)
   do if Eval(caar(x),y) then return(cadar(x),y)
                         else recur(cdr(x),y) ;

Evlis: function (m,a) return (list)
= if null(m) then nil
             else clause
                    x = Eval(car(m),a);
                    y = Evlis(cdr(m),a);
                    return(cons(x,y))
                  end;

Assoc: function (x,a) return (list)
= for (y1,y2):(x,a)
   do case
        null(y2) -> return(nil);
        equal(y1,caar(y2)) -> return(car(y2));
        others -> recur(y1,cdr(y2))
      end;

Pairlis: function (v,e,a) return (list)
= for (x,y,z):(v,e,a)
   do if null(x) then return(z)
                 else clause
                       x' = cdr(x);
                       y' = cdr(y);
                       z' = cons(cons(car(x),car(y)),z);
                       recur(x',y',z')
                      end;
```